Capítulo 3

Tipos y estructuras de datos

Todo programa utiliza variables y constantes para representar cantidades y/o cualidades del problema que implementa. Antes de poder usar una variable en nuestro programa deberemos haberla declarado, para que el compilador (o intérprete) del lenguaje sepa a qué atenerse. La declaración de una variable le indicá al compilador el nombre con el cual nos referiremos a ella, su ámbito deavida y visibilidad y el tipo de datos asociado a la misma.

Nombrado

Puede que el lenguaje restrinja el nombrado de las variables que, por otro lado y como ya hemos visto, habrá de ser suficientemente explícito y reflejar los términos abstractos del problema. Concretamente, los compiladores no permitirán declarar variables cuyo nombre coincida con alguna de las palabras reservadas del lenguaje.

Ámbito de vida

El ámbito de vida o de una variable determina durante cuánto tiempo existe. Una variable declarada como global a un módulo o al programa completo existirá durante todo el tiempo de ejecución del mismo. Una variable local a (o un parámetro formal de) una función existe sólo durante la ejecución de dicha función.

Visibilidad

La visibilidad de una variable determina desde qué puntos de nuestro código podemos hacer referencia a ella.

Las variables globales al programa pueden ser refereidas desde cualquier sitio. Las variables globales a un módulo del programa pueden ser refereidas sólo desde dicho módulo.

Las variables locales a (o parámetros formales de) una función sólo pueden ser referidas desde el código de dicha función.

Si declaramos una variable local con el mismo nombre que una global, la global quedará oculta durante la ejecución de dicha función.

Tipo

El programador escogerá el tipo de datos de la variable en función de la naturaleza del concepto que representa y del conjunto de posibles valores que queremos que la variable pueda tomar. El tipo de una variable podrá ser uno de los tipos básicos que el lenguaje conozca de antemano, o bien de un tipo derivado de estos.

Los tipos numéricos básicos determinan posibles valores dentro de un rango. Por ejemplo, una variable entera no podrá contener (y por lo tanto no podrá valer) el valor decimal 3.5. Una variable declarada como siempre positiva, no podrá contener un valor negativo.

Valor y formato

Es preciso que quede clara la diferencia entre el valor que una variable tiene (o contiene) y el formato o manera en que dicho valor puede ser representado.

Recuerde siempre.

> Un Kilo de oro tiene un determinado valor, independiente de su formato: lingote, polvo,...

Durante la explicación que sigue veremos ejemplos de un mismo valor numérico representado en diferentes formatos (bases de representación).

Tipos básicos

En este apartado presentaremos brevemente los tipos básicos de datos para poder comprender mejor las denominadas estructuras dinámicas de datos que veremos a continuación.

Como parte de los ejercicios prácticos que realizaremos más avanzado el curso, se manejarán estos tipos de datos y las estructuras dinámicas de datos, por lo cual aquí no abundaremos mucho en el tema.

Recuerde que el tipo de una variable o contante determina la naturaleza y cantidad de valores distintos que puede tomar.

Normalmente, para cada tipo básico, el lenguaje ofrece una forma de expresar contantes de dicho tipo.

Sin tipo

Si queremos declarar una función que no devuelve nada, esto es, un procedimiento, usaremos el pseudotipo `void`.

Por lo demás, no pueden declararse variables de este pseudotipo.

BIEN

```
void esperar(unsigned segundos);
```

MAL

```
void variable_nula;
```

Lógico

En muchos lenguajes existe un tipo de datos lógico (*boolean*) capaz de representar los valores lógicos CIERTO o FALSO. El resultado de las operaciones lógicas (comparación)

y de su combinación mediante operadores lógicos (AND, OR, NOT, etc.) es un valor de tipo lógico.

En C y en C++ no existe un tipo de datos lógico, sino que cualquier expresión puede ser evaluada como una expresión lógica. El valor numérico cero equivale al valor lógico FALSO, y como negación, cualquier valor numérico distinto de cero es equivalente a un valor lógico CIERTO.

Carácter

Representan caracteres ASCII o ASCII extendido (Latin1).

- ASCII: `'\0'`, `'a'`, `'B'`, `'?'`, `'\n'`
- Latin1: `'á'`, `'Ü'`

```
char letra;
```

Entero

Son números positivos o negativos o sólo positivos, de naturaleza entera, esto es, que no pueden dividirse en trozos (sin decimales o con parte decimal implícitamente 0). Sirven para numerar cosas o contar cantidades.

Los valores enteros pueden ser representados en diferentes bases o formatos.

- En decimal (base 10): `0, -1, 4095, 65184`
- En octal (base 8): `00, -01, 07777, 0177240`
- En hexadecimal (base 16): `0x0, -0x1, 0xFFF, 0xFea0`

```
int saldo_bancario; /* Nºs negros o rojos */
```

```
unsigned numero_de_dedos; /* Jugando a chinos */
```

Real

Son números positivos o negativos con o sin decimales. Son los denominados "coma flotante".

- Notación punto: `3.14159265358979323846`
- Notación Exponente: `42E-11` que vale $42 * 10^{-11}$

```
float distancia, angulo;
```

```
double PI;
```

Enumerado

Para variables que pueden tomar cualquiera de los valores simbólicos que se enumeran.

Internamente se representarán como un valor entero.

```
enum {
        pulgar,
        indice,
        corazon,
        anular,
        menique
} dedos;
enum {varon, hembra} sexo;
dedo = anular;
sexo = hembra;
```

Puntero

Son variables que, en vez de contener un dato, contienen la dirección de otra variable del tipo indicado.

Se declaran con el caracter `*`. Con el operador `&` se puede obtener la dirección de una variable. Para *de-referenciar* un puntero y así acceder al valor de la variable a la que apunta se utiliza el operador `*`.

```
int variable = 7;
int * puntero_a_entero;
puntero_a_entero = & variable;
*puntero_a_entero = 8;
/* Ahora variable vale 8 */
```

Como veremos dentro de poco, las variables de tipo puntero son la base para la creación de estructuras dinámicas de datos.

La dirección de valor 0 (cero) es tratada como una marca especial. El simbolo NULL asignado a una variable de tipo puntero, le asigna un valor cero, que com odirección es inválida (no puedeser de-referenciada), pero que nos sirve para conocer que un puntero "no apunta" de momento a ningún sitio.

Agrupaciones de datos

Vectores

Son agrupaciones de información homogénea. Se declaran indicando el tipo de cada componente, su nombre y, entre corchetes (`[]`) su dimensión o tamaño. Pueden ser de una dimensión o de varias, en cuyo caso hablamos de matrices.

Los contenidos de un vector o matriz se almacenan de forma contígua en memoria. Sus contenidos son accedidos indexando (con `[`*indice*`]`) con otro valor entero como índice.

En C la indexación comienza por cero.

```
char buffer[80];
float tabla[100];
unsigned char screen[1024][768];
unsigned char icono[16][16];
buffer[0] = 'a';
tabla[100] = 1.0; /* FUERA */
```

Las tiras de caracteres son también vectores. Una tira de caracteres constante se expresa entre comillas dobles. Su último caracter es implícitamente un caracter nulo ('\0').

```
char Hello[12] = "Hola Mundo!";
Hello[0] = 'M';
Hello[6] = 'a';
Hello[11] == '\0'; /* Fin */
```

Mover un icono por pantalla en blanco.

Estructuras

Agrupan en una única entidad un conjunto heterogéneo de campos relacionados que conforman una información concreta.

Los campos se acceden con la notación *.nombre_de_campo*.

En C se pueden definir nuevos tipos de datos con la sentencia `typedef`.

alumno.h

```
typedef struct Alumno {
    char * Nombre, * Apellido;
    struct Fecha {
        int Anyo, Mes, Dia;
    } Nacimiento;
    float Edad;
} Alumno_t;
```

alumno.c

```
#include "alumno.h"
Alumno_t yo;
Alumno_t Clase[22], *actual;
yo.Edad = 33;
actual = &Clase[0];
*actual.Nombre = "Ramon";
actual->Apellido = "Ramirez";
```

Estructuras dinámicas de datos

Cuando nuestros programas necesitan manejar agrupaciones honogéneas de datos cuya cantidad desconocemos de antemano, organizaremos estos como una estructura dinámica de datos.

Ejemplo: Imagine un programa para ordenar alfabeticamente las palabras contenidas en un fichero. No sabemos a priori el número de palabras que puede haber: 100, 1000, cienmil o un millón. El programa debería ser capaz de adaptarse en cada caso, y ocupar tan sólo la memoria necesaria.

Una estructura dinámica de datos es una estructura de datos (algo capaz de contener datos) que puede crecer y encoger en tiempo de ejecución. Para ello, se construyen mediante otras estructuras de datos que incorporan campos "puntero" para interconectarse entre sí. El valor nulo de estos punteros servirá para marcar los *extremos* de la estructura.

Las estructuras dinámicas, como su nombre indica, se ubican dinámicamente en un espacio de memoria especial denominado, así mismo, memoria dinámica. Esta memoria se gestiona mediante primitivas básicas, cuyo nombre cambia según el lenguaje, pero que coinciden en su funcionalidad, a saber:

- **Ubicación de un número de bytes:** Reservan la cantidad de memoria dinámica solicitada. Devuelven la dirección de dicha memoria. `malloc` (en C) o `new` (en C++).
- **Liberación de espacio ubicado:** Liberan la zona de memoria dinámica indicada, que debe haber sido previamente devuelta por la primitiva de ubicación. `free`, (en C) o `delete` (en C++).

Si queremos usar una estructura dinámica de datos en un programa, deberemos disponer de (o programar nosotros), un conjunto de funciones de utilidad que sean capaces de realizar las operaciones básicas sobre dicho tipo: insertar y extraer un elemento, buscar por contenido, ordenar, etc., etc.

Entrar en detalles sobre las estructuras dinámicas de datos daría para un libro entero. En los capítulos posteriores se tendrá oportunidad de hacer uso de ellas y conocerlas más a fondo. En este capítulo nos limitaremos a comentar ciertas características básicas de las más comunes, que son:

Listas Son una estructura lineal de elementos. Como los eslabones de una cadena.

Según como cada elemento esté vinculado al anterior o/y al posterior hablaremos de:

- **Lista simplemente encadenada:** Cada elemento apunta al siguiente. El puntero "siguiente" del último elemento será nulo. Solo permite el recorrido de la lista en un sentido y limita las operaciones de inserción y extracción.

  Como referencia a la lista mantendremos un puntero al primero de sus elementos. Estará vacia si este puntero es nulo.
- **Lista doblemente encadenada:** Cada elemento apunta tanto al siguiente como al anterior. El puntero "siguiente" del último elemento y el "anterior" del pripero, serán nulos. Permite el recorrido de la lista en ambos sentidos y operaciones de inserción/extracción de cualquier posición/elemento de la misma.

  Para manejar la lista mantendremos un puntero al primero de sus elementos, o bien otra estructura que guarde punteros a los dos extremos. Estará vacia si este puntero (o ambos) es nulo.

Según el orden de inserción/extracción de elementos hablaremos de:

- **Cola FIFO:** El primero en entrar será el primero en salir.
- **Pila LIFO:** El último en entrar será el primero en salir.

Árboles Son una estructura arborescente de **nodos**. Todo arbol tendrá un **nodo raíz**, y los nodos que no tengan descendientes son denominados **hojas**.

Según el número de hijos de cada nodo hablaremos de:

- **Árbol binario:** Cada nodo tiene dos hijos.
- **Árbol N-ario:** Múltiples hijos por nodo.

Tablas Hash Son estruturas mixtas que, normalmente, se componen de un vector (de tamaño fijo) de punteros a una estructura dinámica. Su uso fundamental es la optimización de la búsqueda de datos, cuando estos son muy numerosos.

Toda tabla *hash* utiliza una función de barajado (función *hash*) que, en función de un campo clave de cada elemento, determina la entrada del vector en la que debería encontrarse dicho elemento.